Security analysis of Android applications

Bertrand Bonnefoy-Claudet

Research Master Thesis

Research Master Thesis

Security analysis of Android applications

Author:
Bertrand Bonnefoy-Claudet

Supervisors:
Thomas Jensen
David Pichardie
Celtique

Supélec

Abstract

The now ubiquitous Android platform lacks security features that are considered to be necessary given how easily an application can be uploaded on markets by third-party developers and distributed to a large set of devices. Fortunately, static analysis can help developers, markets and users improve the quality and security of applications at a reasonable cost by being automated. While most existing analyses target specific security properties, we take a step back to build better foundations for the analysis of Android applications. We describe a model and give semantics for a significant part of the system by studying what obstacles existing analyses have faced. We then adapt a classical analysis, known as points-to analysis, to applications. This leads us to design and implement a new form of context-sensitivity for Android, paving the way for further experimentation and more specific security analyses.

Contents

Introduction

Android[1] is an operating system designed for mobile devices such as smartphones and tablets. It provides a lot of features for users but most notably empowers any developer to write applications and thus extend its feature set. As it becomes widespread and risks increase, its security is more and more studied.

The fact that Android applications can come from third-parties and are used for many tasks makes Android, the system and its applications, a target for malicious code. Security risks include privacy leak and hijacking. For example, geographic location and contacts can be considered to be private and should not leak to unauthorized third-parties. The complexity of the application framework can lead developers to make mistakes in their code, such as making incorrect assumptions about security properties or using dangerous libraries. Vulnerable applications are an easy target for malicious ones. By contrast, the operating system is harder to attack.

Naturally, the application framework comes with security features aimed at restricting what applications can do. This is done with a permission system and this gives users an idea of the overall security policy. However, permissions cannot ensure all of the security properties. Consequently, there are security guarantees users cannot have from the system.

As a user cannot trust every developer, a promising approach is the automated analysis of applications to derive security properties from them. Analysis of programs is a well-studied problem but Android introduces new challenges for analyzers. Applications are written in Java but their execution is highly dependent on the Android framework, which makes it hard to infer properties without being based on a model of the underlying layers. Such a model must match the actual system closely enough to be realistic, especially if real-world applications are to be analyzed.

The first section of this report is more technical and describes, in a concise manner, the platform applications rely on and its security mechanisms. In section 2, we review existing automated analyses aimed at tackling the aforementioned issues. Section 3 provides a formalization of Android which improves on existing work. Section 4 presents a novel approach to analyzing Android applications, and an implementation, based on the formal model from the previous section.

1 Security on Android

In a nutshell, Android is based on a modified Linux kernel and comes with libraries and an application framework that enable various applications mainly programmed in Java to each run in a controlled manner.

1.1 Platform

1.1.1 Linux

The Linux kernel is the basis for Android and its security. Historically, there have been a significant number of changes from the mainline kernel that are specific to the mobile-world, such as energy-

[1] `http://www.android.com/`

saving scheduling and memory management. However, since recent Linux versions, most of these changes (most notably, wakelocks and autosleep) have been integrated into the mainline kernel[2].

Linux provides a security mechanism called discretionary access control (DAC). It assigns a user ID (UID), group ID (GID) to every process and file and defines permissions regarding operations that processes can perform on files. Discretionary means that users can affect the security policy. In Linux, the owner of some file (a process whose UID is the same as the file's UID) can change its security attributes (read, write and execute) for three classes of users (owner, group and others). This is better described in [1, chapter 10].

As of Android 4.4, the system also runs with a mandatory access control (MAC) mechanism named SELinux, in enforcing mode[3].

1.1.2 Applications

Applications are the main building blocks of an Android system. Some of them are system applications and are pre-installed whereas others are applications developed by third-parties.

There are two main filesystems on a device's internal storage, *system* and *data*. The former is read-only and for pre-installed applications, whereas the latter contains other applications and their files, with proper Linux permissions such that no other application can access them.

An application comes in an *.apk* file. It always contains an *AndroidManifest.xml* file, that we will later refer to as the application's manifest, ressources and a *.dex* file which form the executable. Optionally, an application can contain native libraries in the form of *.so* files. On installation, each application is assigned its unique UID and GID, and permissions declared in the manifest.

Loading an application is done by forking a system process called Zygote[4]. This is a lightweight Linux mechanism [1, chapter 3] and this allows for memory sharing which is crucial for low-memory devices (for instance, the framework's code and static data are shared across applications). The child process then executes the Dalvik virtual machine (VM), which runs the application's *.dex* file[5]. The architecture is schematized in figure 1.

To isolate applications from each other, Android leverages DAC to put each of them in a sandbox. When a process forks from Zygote, it is assigned the unique UID and GID of the application it instanciates. This guarantees that each application has its own virtual memory and private storage space, resulting in a sandbox.

1.1.3 Application framework

Java is the main programming language for applications. Android provides developers with a library and a Software Development Kit (SDK) to help them build, manage and test their applications.

The Android library is included in every application and an implementation of the Application Programming Interface (API) is available on every device [2]. It facilitates the interaction between

[2] `http://lwn.net/Articles/514901/`
[3] `https://source.android.com/devices/tech/security/enhancements44.html`
[4] `http://developer.android.com/training/articles/memory.html`
[5] `http://source.android.com/devices/tech/dalvik/`

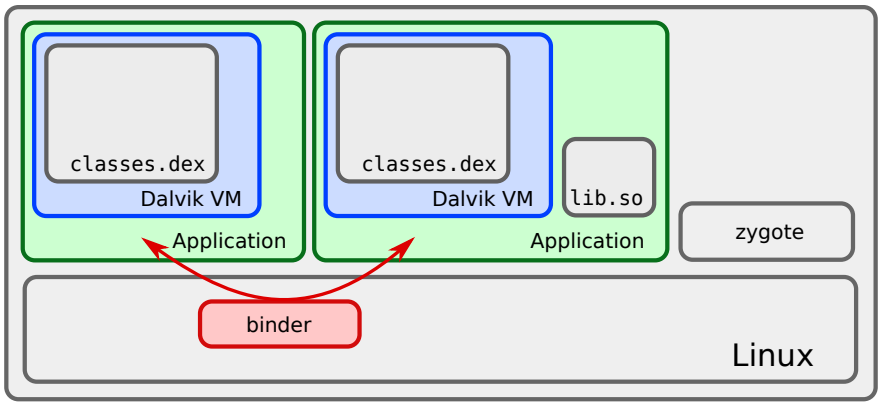

Figure 1: Simplified architecture of Android. Communication between applications is enabled by the binder driver.

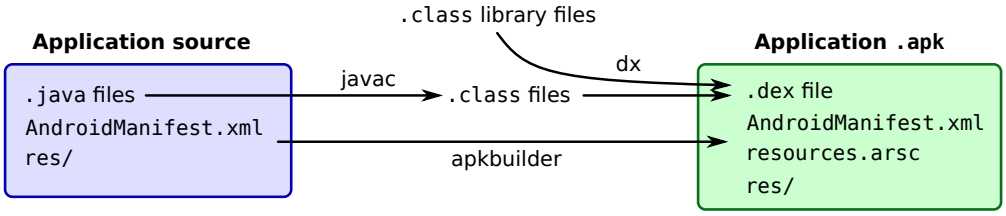

Figure 2: Android's simplified toolchain. Java source code is compiled to *.class* files which in turn are merged into a Dalvik executable. Some resources are compiled while others remain in the *res/* folder.

an application and the rest of the system.

Java source code of the application is compiled to Dalvik bytecode into the *.dex* file. See figure 2 or the documentation[6] for more details. Although unusual, it is possible to split an application into several *.dex* files and load them dynamically. As mentioned previously, a developer can include a native library to her application, for example to perform computationally intensive operations. This can be done with the Native Development Kit (NDK) and uses the Java Native Interface (JNI).

Once packaged in an *.apk* file and signed, an application can be distributed via Google Play (formerly the Android Market).

1.1.4 Dalvik bytecode

The *.dex* file of an application contains Dalvik bytecode is interpreted by an instance of the Dalvik VM. The format of the executable and the bytecode specification are described at length in the official documentation. They shares similar concepts with Java [3].

[6] `http://developer.android.com/tools/building/`

```
public double m2(int a)
{
  if (a != 0)
    return 1.0;
  else
    return 2.5;
}
```

(a) Java source code

```
public double m2(int);
0: iload_1
1: ifeq 6
4: dconst_1
5: dreturn
6: ldc2_w #double 2.5d
9: dreturn
```

(b) Java bytecode

```
public double m2(int);
0: if-eqz v3, 5 // +5
2: const-wide/high16 v0, #long 4607182...
4: return-wide v0
5: const-wide/high16 v0, #long 4612811...
7: goto 4 // -3
```

(c) Dalvik bytecode

Figure 3: Example of a compilation from Java source code to Dalvik bytecode (dexdump output). It illustrates the use of the stack in Java bytecode and virtual registers in Dalvik bytecode.

Classes are assembled in the *.dex* file instead of being in separate *.class* files. Furthermore, the classes' constant pools are merged into one pool in this executable, which saves space on mobile devices.

The Dalvik VM is register-based whereas the Java VM is stack-based. It has been specifically designed for low-memory devices with limited computational capabilities. The bytecode [3] counts 257 instructions and many instructions formats (for different kinds of arguments and encoding). Figure 3 shows a code sample from [4] at different compilation stages. The type system is also different as it is more ambiguous in Dalvik. For example, an instruction that operates on a 32-bit int can operate on a 32-bit float as well and the integer constant zero is also treated as a null reference.

1.1.5 Components

Applications are divided into well-specified components. Each component inherits from one of the following classes, depending on its role within the application:

- Activity: user-interface
- Service: background process
- BroadcastReceiver: reaction to events
- ContentProvider: persistent storage

Each component has its own lifecycle and provides methods that will be called when its services are requested. The application's manifest determines which components are active and accessible and the lifecycle describes how and when a component's methods are called by the framework.

1.1.6 Inter-component communication

As a result of the application sandbox, there is a need for an inter-process communication (IPC) mechanism to enable applications to communicate with each other, especially with system applications. Actually, communication is done at the component level. This means that communication between two components is done the same way no matter whether they are part of the same application or not. As in [5], we will refer to it as inter-component communication (ICC).

ICC in Android is enabled by the Binder driver. Along with the API, it is aimed at making inter-component communication seamless. However, applications need proper permissions depending on the messages they want to send. These permissions are declared statically by developers in each application's manifest.

The following explanations are inspired from [6]. The main message object used for communication is the Intent. It can have different meanings such as an action to be performed or a notification. Examples of the use of Intents include:

- a component calling an activity's method
- a system component notifying selected broadcast receivers of an event
- a component using a remote service component as a local object

Intents can be either explicit or implicit. Explicit Intents have the destination unambiguously specified, that is, they will only reach the expected component. On the other hand, implicit ones just declare an action to be done. If the destination is an Activity, then the user will be shown a user interface with suitables activities to choose among. If the destination is a Service, then the system will randomly choose any one that is suitable to perform the action. Otherwise, if the destination is a BroadcastReceiver, the Intent will reach any receiver that has registered for this kind of Intent.

Other passed objects can be Uris (for addressing values in ContentProviders), Bundles (generic containers often passed along an Intent to start an Activity), Cursors (references to query results) and ContentValues (containers for ContentResolvers).

The Android communication model is generic. It only consists in components communicating via Binder and restricted by permissions. Applications are thus enabled to use other applications' services. This encourages code and ressources reuse, which benefits portable devices Android has been built for.

Classical Linux IPC mechanisms, as listed in [1, chapter 5], are available to applications but those are hardly adapted to Android applications because of process isolation. Using them is discouraged, partly for security reasons [7].

1.2 Security mechanisms

There are numerous ways of attacking an Android-powered phone. As any computing machine, it has vulnerabilities which reveal flaws in the system, with threats ranging from physical attacks to browser exploits. However, there is a great concern for application security. Almost all applications are distributed via Google Play. Unfortunately, this means that devices execute untrusted code [7].

Furthermore, current security mechanisms provided by Android cannot mitigate all attacks. This trivially follows from the fact that some applications manipulate sensitive data or perform sensitive operations and that the Android environment does nothing to prevent them from misbehaving. On the other hand, the system and correct applications should at least be able to protect themselves.

We will only consider attacks leveraging applications. Although there are many external attacks, both physical and remote [8], they are less prone to an automated security analysis.

[7] `http://source.android.com/devices/tech/security/`

1.2.1 Threat model

Misbehaving applications can either be malicious or vulnerable. Malicious applications perform actions they are not expected to do. They can only be curbed to a certain extent by the Android security model. On the other hand, vulnerable applications have good intentions but do not protect themselves properly against other applications or external threats.

We thus consider application code to be potentially dangerous and threats to come from one or more applications.

1.2.2 Android permissions

Beyond the application sandbox, Android permission mechanism governs most of ICC in the system.

Each application defines several sets of permissions in its manifest. It does this in two ways: permissions it wants to be granted and, for each of its components, permissions required to access the component. A permission has a level: normal, dangerous, signature or signatureOrSystem, each of which determines how hard it is to obtain the permission.

At install-time, the user is shown the permissions requested by the application. However, permissions the application requires to protect its components are not displayed. Only permissions with the normal level can be granted this way.

The system is responsible for checking permissions but additional checks can be performed by API code called from the application although those do not increase security as a developer can bypass it.

Some of the permissions are solely handled by the Linux DAC [2]. For example, applications requesting the `INTERNET` permission have their UID put in the `inet` group. This is also the case for writing on the SD Card with `WRITE_EXTERNAL_STORAGE` and using Bluetooth with `BLUETOOTH`.

1.2.3 Mandatory access control

As of Android 4.4, SELinux is integrated in the system to circumvent some of the limitations of the Linux DAC. The original system does indeed not enforce any restrictions on root processes and applications.

Processes like Zygote are highly privileged. Therefore, compromising such a process boils down to compromising the whole system. SELinux addresses this issue and constrains processes in order to prevent them from doing anything they are not expected to do. This is done by specifying policy files for the root domain[8].

While applications are isolated by default, mistakes in their code can lead them to be vulnerable (see section 1.3). Unfortunately, the Dalvik VM does not provide security, as native components bypass it. SELinux can constrain whole applications but it would require application-specific policies for this kind of protection to be effective. It is still unclear how SELinux has been integrated into Android and what plans developers have for securing applications.

[8] `http://source.android.com/devices/tech/security/se-linux.html`

1.3 Security of applications

Security issues in Android applications have been investigated in several papers. In particular, Enck et al. [3] analyze 1,100 free Android applications and find that the most common issues are leaks of phone identifiers and physical locations. We focus on the fact that application code is untrusted.

1.3.1 Overprivileged applications

Some applications are overprivileged [2]. Such applications request one or more permissions they do not need. They make potential vulnerabilities more critical and are more likely to be malicious.

Declaring enough permissions is essential for developers as Android kills any application issuing an API call without having permission to do so. Moreover, adding a declared permission along a new version of an application implies that it will need users' consent to be updated on devices where it has been installed. This incentivizes developers to request permissions they might need in the future.

On the other hand, requesting unused permissions is not as critical, still in a developers' point of view, as the only disadvantage is to have some users reject their application at install-time for they may look suspicious.

One main reason for overprivilege determined by Felt et al. [2] is the fact that the API is not well documented enough and that it can be hard to infer permissions required by a method from its description in the API documentation.

1.3.2 Insecure applications

To benefit from the sandbox, an application must take precautions and be careful regarding how it stores its data and uses ICC.

Applications must ensure that data they store cannot be used by unauthorized components by assigning them proper Linux permissions. It is their responsibilty because of the DAC model. For instance, the Skype application used to let sensitive data stored in its storage space be readable and writable by any other application [9].

As components interact with each other via ICC, they must ensure that data they send cannot be read by unauthorized receivers. Conversely, they must only react to expected messages, not spurious ones. There are indeed a few caveats a developer must be aware of when writing communication code. In particular, implicit Intents are dangerous when used for Services or BroadcastReceivers. Uses of implicit Intents for a Service can lead to Service hijacking and uses of these for BroadcastReceivers can lead to Broadcast theft [6].

1.3.3 Malicious applications

Applications with evil intentions can take advantage of three kind of flaws.

System Vulnerabilities have been found in system processes. They are highly privileged and thus many assets are at stake, including the user private information already there or obtained through hijacking. Attacks on the system are often performed to "root" the device, which means to the user that he will then be able to let some applications gain root privileges (which is initially not the case in most of the sold devices). *GingerBreak* is a well known exploit used to root phones running Android 2.3 also known as Gingerbread. Using MAC such as with SELinux, most of these attacks could be mitigated [9].

Insecure applications A malicious application can leverage an insecure application to perform unauthorized actions. Potential vulnerabilities are described in section 1.3.2.

Permissions Android permissions are too coarse-grained to cover all security issues. Therefore, some applications *have to* require more permissions than what they would need with finer-grained permissions (they are not overprivileged in the previous sense as there is no smaller suitable permission set). A common issue is an information flow that breaks privacy rules despite minimal permissions in the manifest [3, 10, 11, 12].

For example, an email client could need access to both the network (`INTERNET`) and the user's contacts (`READ_CONTACTS`). Network access is required to fetch and send emails whereas reading contacts is useful to associate email addresses with names. However, given these two permissions, there is no guarantee that the application will not send contact information to a remote server, resulting in a privacy leak.

1.4 Conclusion

Applications are one of the main threats in an Android system. Our focus will be on the previously described overprivileged, insecure and malicious applications. As pointed out, a malicious application can leverage an insecure one to perform its mischief. Issues can thus come from more than one applications.

2 Automated analysis

The need for automated analyses arises in malware detection, quality assessment and bug finding. The huge number of applications that are developed and uploaded to online markets makes manual techniques of reverse engineering and code auditing impractical. Analyses can help reduce the cost and time of such tasks and sometimes even be fully automated. On the developer side, the lack of time and the complexity of the system and the API can cause bugs in applications, reducing their performance or their security. Assisting the developement of Android applications with analyzers can reduce this risk. There are tools such as lint[9] and the monitor[10] which are already provided in the Android SDK.

[9] https://developer.android.com/tools/help/lint.html
[10] https://developer.android.com/tools/help/monitor.html

2.1 Considerations on analyzers

2.1.1 Ahead-of-time, online and offline

An application analyzed before it is run by a user is said to be analyzed *ahead-of-time*. When it comes to malware analysis, the main question is whether to perform it on a user's phone or not. The former is said to be *online* and the latter *offline*. An online analysis has the advantage of knowing the full execution enviroment. An offline analysis is also ahead-of-time.

2.1.2 Soundness and precision

The goal of an ahead-of-time analysis is to determine properties about potential behaviour of an application. A sound analysis considers all possible executions and can thus guarantee the absence of some of them. It is said to be unsound otherwise. The more applications an analysis is able to state properties on, the more it is precise.

In the case of classifying applications into the *good* and the *bad* ones, we usually talk about alarms. When an analyzer says of a good application that it is bad, it raises a false alarm, or false positive. On the other hand, when it says of a bad application that it is good, it is said to output a false negative. A sound analyzer never outputs false negatives, and the fewer false alarms an analyzer raises, the more precise it is.

Unfortunately, many interesting properties such as "the application cannot leak private data" or "the application is not vulnerable to intent spoofing attacks" are not decidable. This means that given an analyzer and such a predicate on applications, there will always exist an application for which the analyzer will not be able to prove nor refute the predicate. In other words, a sound analyzer will always raise at least one false alarm for some application.

We must thus expect sound and interesting ahead-of-time analyses to lack precision to a certain extent. By contrast, analyses that are both dynamic and online can make stronger guarantees, as they can monitor everything that is happening, but often come with a run-time overhead noticeable by a user.

2.1.3 Static and dynamic

Static and dynamic analyses are the two main families of program analysis. The former works at compile-time on the code of an application and without executing it, whereas the latter analyzes an application during its execution.

Static analysis can be sound because it can consider all possible execution paths and never give false negatives. However, to achieve that, as seen in section 2.1.2, it must over-approximate the set of potential executions and thus necessarily raises false alarms on some applications. Moreover, both under- and over-approximations can be needed to deal with the complexity of some systems, especially for Android as we will see in further sections, and respectively lead to false negatives and positives.

Dynamic analysis follows only a few real executions closely and can hence be more precise. For instance, there is no need to over-approximate the control flow as is usually the case with static

```
void doLeak() {
  int x = 17 + source()
  sink(x)
}
```

(a) Tainted variable x is leaked.

$$\frac{\text{taint}(y) \cup \text{taint}(z) \subseteq \text{taint}(x)}{\texttt{x = y + z}}$$

(b) Rule propagating taints through addition and assignment. If y or z is tainted, then x is also tainted.

Figure 4: Example of taint analysis

analysis. Unfortunately, it cannot be sound when done ahead-of-time because it cannot consider all possible execution paths. However, it can still make strong guarantees when performed online but then cause a runtime overhead. This overhead can be reduced with approximations, making the analysis unsound or less precise.

The main tradeoffs between static and dynamic analyses are precision, soundness and cost. These characteristics have a different impact depending on whether the analysis is to be performed online, ahead-of-time or offline.

2.2 Taint analysis

The main security property targeted by existing analyses on Android is information-flow security. It consists in detecting flows leaking information from secure locations to less secure locations, that is, any observable behaviour revealing information about secret data. In practice, most analyses on Android only focus on leaks due to variable or object assignment, known as explicit flows. They ignore implicit flows which makes them unsound with regard to information-flow security.

Security levels for data sources and variables can be represented by a lattice [13] or just a partially ordered set [10], although usually, two levels suffice: *sources* and *sinks*. Sources in Android are API methods that return sensitive information and sinks are ones that make it leak. Examples of sources are `GsmCellLocation.getCellLocation()` and `TelephonyManager.getDeviceId()`. As for sinks, `SmsManager.sendTextMessage()` and `FileUtils.stringToFile(String, String)`. Rasthofer et al. [14] classify most API methods into sources, sinks and benign methods.

Explicit flows are usually detected with a technique known as taint analysis [10, 11, 15]. It consists in propagating *taints* from sources to sinks through operations on values and assignments. An example of taint anlysis is shown in figure 4.

When taint analysis is static and sound, over-approximations are unavoidable (cf. section 2.1.2). We give here two examples of common approximations. First, when two branches merge, as shown in figure 5a, a reasonable analyzer considers an approximate state at the junction point. Second, on array assignment, as in figure 5b, it is usual for analyzers to not track array indexes and taint whole arrays. Even when array indexes can always be determined, which is the case with dynamic analysis, such an approximation can be made to decrease the analysis overhead [12].

```
void joinBranches() {
  int x = 0;
  if (foo()) {
    x = source();
  }
  sink(x);
}
```

(a) If the analyzer does not know what `foo()` can return, then it will consider x to be tainted when it is leaked through `sink`().

```
void assignArray() {
  int[] a = new int[10];
  a[foo()] = source();
  x = a[0];
  sink(x);
}
```

(b) Analyzers usually treat an array as a whole for perfomance reasons and therefore taint the whole array and x even if they know that `foo()` cannot return 0.

Figure 5: Approximations with taint analysis

2.3 Other security analyses

Felt et al. [2] designed Stowaway to analyze applications and detect overprivilege in them. The analysis is performed in two steps. First, a permission map is built to associate Android permissions with each method of the API. Applications are then statically analyzed to determine which of the API methods they call and, with the permission map, what permissions they really need. An application is overprivileged if it declares unneeded permissions in its manifest (as characterized in section 1.3.1).

Several tools [6, 5] look for vulnerabilities in Android applications (described in section 1.3.2) using static analysis on Dalvik bytecode. There are many different potential vulnerabilities in applications such as activity and service hijacking, enabled by Intent- and broadcast-based attacks.

2.4 Challenges posed by Java and Android

2.4.1 Reflection

Reflection is enabled by Java and consists in examining and modifying classes and methods at runtime using, among others, iterators and strings. A typical pattern taken from [16] is shown below. It makes static analysis difficult because object classes, and thus targets of calls, can then be arbitrarily hard to predict.

```
String className = ...;
Class c = Class.forName(className);
Object o = c.newInstance();
T t = (T) o;
```

If some class Foo is available, it can be referenced as Foo.class instead of Class.forName("Foo"). This is often used in Android to build intents:

```
Intent intent = new Intent(this, DisplayMessageActivity.class);
```

Reflection in Android applications is mostly used for accessing hidden or private API methods, ensuring backward compatibility or generating objects from serialized of user input [17]. Some

applications do not use reflection but libraries they import do, hindering their analysis as well.

2.4.2 Uris

In addition to reflection, Android uses string Uris to reference content in ContentProviders and perform queries. Some of theses strings are built dynamically and determining their value, which is critical for taint analysis, can be challenging. [11] and [10] resolve string prefixes to associate a string with content it points to. Prefixes can be enough because the last characters are often parameters not needed for the analysis. Fortunately, those are often the only dynamic part of the string.

2.4.3 Dalvik bytecode

The differences mentioned in section 1.1.4 have two implications. First, working with Dalvik bytecode is different from working with Java bytecode. Second, getting Java bytecode back from Dalvik bytecode is non-trivial [4]. As a consequence, there are two main approaches to dealing with applications regarding the analyzed language. Some analyses work directly on Dalvik bytecode but they lack all the tools available for analyzing Java. Others choose to first retarget (or decompile) Dalvik bytecode to Java bytecode (or source code) and then use tools such as WALA[11] and Soot[12].

2.4.4 Multi-threading

Application execution is mainly event-based and most API calls are asynchronous. While most of the GUI logic and callbacks run in one main thread, multi-threading is allowed by the Java Thread class and Android wrappers around it[13]. Furthermore, Service components run in a separate thread. This results in potentially thread-unsafe code that can be very hard to analyze because of all the potential interleavings of instructions between threads. To our knowledge, nobody has designed a static analysis sound with regard to multi-threading except Fuchs et al. [10] but it is not clear how precise their analysis is.

2.4.5 Native code

Applications can come with native libraries as *.so* files whose functions are used as Java methods. Analyzing native code is very different from analyzing Java and we have not seen any analysis tackle the difficult task of integrating native code analysis into an Android analysis. In fact, analyses usually either over-approximate native calls [15] or ignore them. For instance, in taint analysis, the safest approximation is to consider all references in arguments and the return value of a native call to be tainted.

[11] `http://wala.sourceforge.net`
[12] `http://www.sable.mcgill.ca/soot/`
[13] `http://developer.android.com/guide/components/processes-and-threads.html`

2.4.6 Android framework and API

Applications are built against both a framework and an API. We call framework the Android system which invokes methods of an application and API the set of library methods an application can use. As seen in section 1.1.5, the behaviour of an application is determined by a manifest, the `AndroidManifest.xml` file, and components. The manifest declares how each of these components can be initialized.

Unlike classical programs with a single *main* function, an application can have multiple entry points, also known as callbacks. The main kind of callbacks are lifecycle callbacks. They are called on components by the system to respond to system events. The execution of an application is therefore an initial state (allowed by the manifest) and a framework trace invoking callbacks defined by the application. The framework thus drives the execution of an application but the latter can influence the former via API calls. This results in a strong coupling between the framework and the API. An analysis must thus understand how they work together in order to make sense out of the code of an application.

One approach could consist in analyzing the code of Android. However, we know of no previous work moving down that road. This probably stems from the the fact that Android is hard to analyze [18]:

- The codebase is huge for an analyzer and there are many interdependencies between classes.
- It heavily uses reflection to allow pratical patterns to be used in application's code.
- Much of the code is related to performance improvements and graphical considerations.
- Several programming languages are used.

As a result, Android needs to be modeled. Existing security analyses all use their own model and only give a rough description of how they approximate it. They all seem to be very permissive about callbacks, allowing many unfeasible paths. While some of them claim that the risk of becoming unsound is too great when trying to be precise about Android [15], others say that precision can be needed to achieve significant results [18, 19]. We believe that analyses that do not model Android precisely use other information, which can be type-related or specific to the analysis, to eliminate some of the unfeasible paths and keep precision to an acceptable level.

2.4.7 Conclusion

Analyses we have come across are summarized in table 1. As we aim for a more fundamental and formal approach, we will start by ignoring many complex features such as multi-threading, native code and Uris, which is also the case of many existing analyses. We will only model the "framework" nature of Android and its reflexivity- and event-driven control flow enabled by Intents and callbacks so that it can be the basis for a static analysis. Although modeling Android correctly is hard, it can be noted that some unsoundness does not prevent an analysis from being helpful for bug finding and vulnerability discovery.

	analysis	scope	source	analyzed	framework and API model	extras	target
FlowDroid [15]	static	app	Dalvik	IR (Soot)	activity lifecycle and source-sink classification of the API, stubs for native code and API	native code	taint analysis
ScanDal [11]	static	app	Java	IR	semantics of part of the API	reflection	taint analysis
SCanDroid [10]	static	system	Java	IR (WALA)	lifecycle of components, Intent and URI	reflection	taint analysis
ComDroid [6]	static	app	Dalvik	Dalvik	Intent		Intent- and component-related vulnerabilities
Stowaway [2]	static	app	Dalvik	Dalvik	Intent and URI		overprivilege
Epicc [5]	static	system	Java	IR	lifecycle of components, Intent		Intent-related vulnerabilities
DidFail [20]	static	system	Java	IR (Soot)	leverages FlowDroid and Epicc		taint analysis and vulnerabilities
SymDroid [18]	static	app	Dalvik	IR	client-oriented specification		symbolic execution
Rountev et Yan [21]	static	app	Java	IR (Soot)	GUI		points-to analysis
AppIntent [22]	static and dynamic	app	Java	IR (Soot)	activity lifecycle and GUI		taint analysis
DroidScope [23]	dynamic	system	Linux and Dalvik VM	execution			taint analysis
TaintDroid [12]	dynamic	system	Dalvik VM	execution			taint analysis

Table 1: Summary of existing analyses. In column *scope*, "system" means that a set of applications and their interactions are analyzed. Column *source* contains the format the application needs to be in to be analyzed and column *analyzed* contains the language on which the analysis is implemented, with Java and Dalvik referring to the bytecode, and Soot and WALA being common analysis tools for Java. Column *target* contains the kind of property the analyzer looks for.

3 Understanding Android

We want a better idea of how applications are executed for two reasons. First, we would like a formal approach to explain how applications are executed by focusing on activities and callback-driven execution. We found that previous formal work on Android models other parts such as interactions between components, which we are not currently interested in. Second, we want our approach to be more general than existing analyses, not specific to taint analysis or vulnerability discovery in particular.

After some time working on modeling Android, we came across two papers that outweighed our work to some extent: an operational semantics for Android activities by Payet et Spoto [19] and a points-to analysis by Rountev et Yan [21]. The model presented in this section therefore builds on part of their work with the following differences. We go further than [19] by describing a model in which listener callbacks can be registered in an activity. However, we do not model the view logic as precisely as in [21], which gives a precise semantics for Views but does not describe formallly how Android executes an application.

3.1 Framework

There are two kinds of methods we care about: lifecycle callbacks and listeners. Both can be written by the developer but lifecycle callbacks are set forth by the framework while listeners only react to what the developer wants them to react to.

3.1.1 Activity lifecycle

An activity is responsible for one part of the GUI of an application. It is assigned part or all of the screen and manages a set of views that defines what it looks like and how the user can interact with it. It is meant to focus on one task and delegate the rest to other activities. While developing a complex application can be done within one activity only, it is strongly discouraged. We will only consider fullscreen activities[14].

The callbacks the developer can override are the following ones, where the activity state is passed as a Bundle (class defined in Android):

- `void <init>()`
- `void onCreate(Bundle)`
- `void onStart()`
- `void onResume()`
- `void onPause()`
- `void onStop()`
- `void onRestart()`
- `void onDestroy()`
- `void onUserLeaveHint()`
- `void onSaveInstanceState(Bundle)`

[14]Multiple visible activities are enabled by Fragments (`https://developer.android.com/reference/android/app/Fragment.html`).

- `void onRestoreInstanceState(Bundle)`
- `void onTitleChanged(CharSequence, int)`
- `void onPostCreate(Bundle)`
- `void onPostResume()`

Depending on the state of the running application, Android invokes the callbacks of its activities. The logic for these calls is complex and very hard to fully grasp. We will thus need a model that subsumes most common behaviour, maybe not all of it if precision is most important. The main use of lifecycle callbacks is to respond to visibility changes and maintain the GUI state between instanciations. How they are called mainly depends on the activity stack and the device's available resources.

When an activity is launched, it is instanciated, pushed to the top of the activity stack and thus about to become visible. The system then calls `onCreate`, `onStart`, `onResume` and others to make that happen. However, the activity from which the new one is launched is about to be hidden (we assume fullscreen activities) and must be paused. This is signaled with `onPause` and `onStop`, among others. The framework gives developers an opportunity to react to those events by extending the lifecycle callbacks[15].

A typical `onCreate` implementation is the following:

```
protected void onCreate(Bundle savedState) {
    super.onCreate(savedState); // restore views' states if needed
    setContentView(R.layout.main); // build views from XML layout and set root view as the activity's
    // ... setup asynchronous loading from database to update views
}
```

The API method `Activity.setContentView` uses a process known as inflation. It consists in iterating over an XML file to instanciate and configure a view and its children. For example, the two following snippets are equivalent.

```
XML in res/layout/main.xml:

<LinearLayout>
  <LinearLayout>
    <EditText
      android:id="edit" />
    <Button
      android:text="Add" />
  </LinearLayout>
  <ListView
    android:id="task_list" />
</LinearLayout>

setContentView(R.layout.main);
```

```
LinearLayout ll0 = new LinearLayout(this);
LinearLayout ll1 = new LinearLayout(this);
EditText et = new EditText(this);
Button b = new Button(this);
ListView lv = new ListView(this);
et.setId("edit");
b.setText("Save");
lv.setId("task_list");
ll1.addView(et);
ll1.addView(b);
ll0.addView(ll1);
ll0.addView(lv);
setContentView(ll0);
```

As can be seen in this example, `View` objects can be either inflated from an XML file or created programmatically, and they have a tree-like structure. Here, the `task_list` `LinearLayout` is empty and will be filled with views depending on values in a database. Such a dynamic part is almost always present in applications.

[15] Almost all overrides must call through to the super class method.

3.1.2 Views

A view is in fact a tree of views. Therefore, when we mention a view, we actually refer to the whole tree rooted at this view. For the active activity, the system initially displays the view that has been set as the activity's view via `setContentView`. Dialogs can then be put in front of this initial tree and hide part of the initial tree of views.

As our analysis only cares about what actions are available to the user, not what can be seen, dialogs can be considered to override the underlying view hierarchy because any view behind a dialog cannot be clicked even though it might still be visible.

3.1.3 Listeners

While lifecycle callbacks enable developers to configure their activities when internal events occur, listeners wait for user-triggered GUI events and respond with their own methods.

A listener can be either an object implementing a specific interface or the activity itself. It is associated to an object, usually a view, from which it listens for events. When an event is triggered for a view, the system calls the corresponding method on the listener with the view passed as the only argument.

In the previous example, the button does nothing when pressed because there are no listeners declared. In order to make it do something, we must define a class implementing the `View.OnClickListener` interface, instanciate an object of this class and add it to the view as an `OnClick` listener. A view can have other kinds of listeners such as `OnHoverListener` and `OnLongClickListener`. Defining a listener is most commonly done with an anonymous class or an attribute in the XML referring to a method of the activity.

3.1.4 API

The classes for the Android framework provide the developer with hundreds of methods that we do not want to analyze. Some of them are more important as they are common and have a significant effect on the control flow. As such, they deserve to be modeled. Most of the calls involved in the GUI have a delayed effect, which means that instructions that follow them will not see any change in the environment.

An activity can launch another one with `startActivity(Intent, Bundle)`, where `Intent` designates the target activity and `Bundle` is a container for various launch options. Its delayed effect is to instanciate and push the target activity on top of the stack when the system regains control. The target activity must be specified in the intent, as a class or a string for an explicit intent and as an action, denoting a set of activities, for an implicit one.

A call to `finish()` on the current activity schedules its removal from the stack. This is often done in combination with `startActivity` when a new activity is meant to replace the current one instead of being pushed onto it.

3.2 Semantics

The previous description of Android activities and View system can be formalized. We build a model using mathematical objects and semantic rules to help us understand how it really works and remove ambiguities. This will be the basis for the upcoming analysis.

Java-like languages are classically modeled with a heap [24] that associates references, or locations, with objects which themselves associate field names with values. Additionally, we augment the classical notion of objects with Android attributes. This translates into the following domains:

$$
\begin{aligned}
\textbf{ActivityState} &= \{\mathsf{uninit}, \mathsf{init}, \mathsf{created}, \mathsf{visible}, \mathsf{active}, \mathsf{stopped}, \mathsf{destroyed}\} \\
\textbf{ActivityInfo} &= \{ \\
&\quad \mathsf{state} : \textbf{ActivityState}, \\
&\quad \mathsf{pending} : \textbf{Reference}^*, \\
&\quad \mathsf{listeners} : \mathcal{P}\,(\textbf{Reference})\,, \\
&\quad \mathsf{finished} : \textbf{Boolean} \\
&\} \\
\textbf{Value} &= \textbf{Reference} \cup \textbf{String} \\
\textbf{Object} &= (\textbf{Field} \to \textbf{Value}) \times \textbf{ActivityInfo} \times \textbf{Class} \\
\textbf{Heap} &= \textbf{Reference} \to \textbf{Object}
\end{aligned}
$$

We can thus write $h(r)(f)$ to refer to field f of the object designated by reference r in heap h. Similarly, we can write $h(r)(\mathsf{state})$ to refer to its Android state attribute if it is a reference to an Activity object. We will give a meaning to those values and **ActivityInfo** attributes later. An object $h(r)$ also has a class given by $\mathrm{Class}\,(\mathrm{r})$.

In addition to the heap, we need a way for Android to keep track of its activity stack. It is a stack of references in the model, Android activities being regular objects:

$$\textbf{ActivityStack} = \textbf{Reference}^*$$

Given a heap h and an activity stack as, we can represent an Android state as $\langle\, h,\, as \,\rangle \in \textbf{State}$.

3.2.1 Semantics of Mini

Next we need a language to write callbacks. Let Mini be a Java-like language with very few statements, no inheritance and the following possible statements:

$$
\begin{aligned}
\textsc{String} &: x = \texttt{"}s\texttt{"} \\
\textsc{New} &: x = \text{new } cl \\
\textsc{Assign} &: x = y \\
\textsc{Get} &: x = y.f \\
\textsc{Set} &: x.f = y \\
\textsc{IfGoto} &: \text{if } (*) \text{ goto } pc \\
\textsc{Call} &: x = y.m(a_1, \ldots, a_n) \\
\textsc{Return} &: \text{return } x
\end{aligned}
$$

A program is defined by methods which are grouped in classes. Each method $m \in \mathbf{Methods}$ has a code $c \in \mathbf{Int} \to \mathbf{Statement}$ which is a finite sequence of statements indexed by integers, starting at 0 and ending at end(c). We do not specify the condition in the IfGoto statement because it will not be necessary for the analysis. A small-step-like semantics $(\cdot \to \cdot) \in \mathbf{MiniState} \to \mathbf{MiniState}$ is given in figure 6. The Lookup function used in the definition of Call returns the code associated with a method name and the runtime class of a receiver object. Identifiers this, arg_i and ret refer to the formal parameters and the return value of a method. States are defined in the following domain:

$$\begin{aligned} \mathbf{Env} &= \mathbf{Variable} \to \mathbf{Value} \\ \mathbf{MiniState} &= \mathbf{Int} \times \mathbf{Env} \times \mathbf{Heap} \end{aligned}$$

3.2.2 Android API

The Call rule of Mini is partially overridden by the following API rules. This means that when the class cl and method m correspond to an API call, then one of the following rules applies instead of Call.

x.startActivity(y) pushes the string pointed to by y to the $\mathsf{pending}$ stack of the activity pointed to by x:

$$[\textsc{StartActivity}]\frac{\begin{array}{c} c(pc) = (x.\text{startActivity}(y)) \\ o'_x = o_x\,[\mathsf{pending} \mapsto l(y) :: p] \quad p = o_x\,(\mathsf{pending}) \quad o_x = h\,(l\,(x)) \quad l(y) \in \mathbf{String} \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc+1, l, h\,[l(x) \mapsto o'_x] \,\rangle}$$

x.finish() sets to true the finish field of the activity pointed to by x:

$$[\textsc{Finish}]\frac{\begin{array}{c} c(pc) = (x.\text{finish}()) \\ o'_x = o_x\,[\mathsf{finished} \mapsto \mathbf{tt}] \quad o_x = h\,(l\,(x)) \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc+1, l, h\,[l(x) \mapsto o'_x] \,\rangle}$$

x.addListener(y) adds the listener pointed to by y to the $\mathsf{listener}$ field of the activity pointed to by x:

$$[\textsc{AddListener}]\frac{\begin{array}{c} c(pc) = (x.\text{addListener}(y)) \\ o'_x = o_x\,[\mathsf{listeners} \mapsto l(y) \cup li] \quad li = o_x\,(\mathsf{listeners}) \quad o_x = h\,(l\,(x)) \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc+1, l, h\,[l(x) \mapsto o'_x] \,\rangle}$$

x.setListener(y) has the same effect but also overwrites the attribute, removing previous listeners from the activity:

$$[\textsc{SetListener}]\frac{\begin{array}{c} c(pc) = (x.\text{setListener}(y)) \\ o'_x = o_x\,[\mathsf{listeners} \mapsto l(y)] \quad o_x = h\,(l\,(x)) \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc+1, l, h\,[l(x) \mapsto o'_x] \,\rangle}$$

This API roughly corresponds to the one described in section 3.1.4. The View model is not included in the present model but could be easily added as it is defined in [21]. Instead we just register listeners to activities, which are to be seen as GUI buttons with their respective callbacks. Also, this representation represents Intents as strings and therefore assumes that their targets are explicit and determined statically.

$$[\textsc{String}]\frac{c(pc) = (x = \texttt{"}s\texttt{"})}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l\,[x \mapsto s]\,, h \,\rangle}$$

$$[\textsc{New}]\frac{\begin{array}{c} c(pc) = (x = \text{new } cl) \\ r_x \notin \text{dom}\,(h) \quad cl \in \textbf{Classes} \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l\,[x \mapsto X]\,, h\,[r_x \mapsto []_{cl}] \,\rangle}$$

$$[\textsc{Assign}]\frac{c(pc) = (x = y)}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l\,[x \mapsto l\,(y)]\,, h \,\rangle}$$

$$[\textsc{Set}]\frac{\begin{array}{c} c(pc) = (x.f = y) \\ o'_x = o_x\,[f \mapsto l(y)] \quad o_x = h(r_x) \quad r_x = l(x) \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l, h\,[r_x \mapsto o'_x] \,\rangle}$$

$$[\textsc{Get}]\frac{c(pc) = (x = y.f) \quad r_y = l(y)}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l\,[x \mapsto h(r_y)(f)]\,, h \,\rangle}$$

$$[\textsc{IfGotoTrue}]\frac{c(pc) = (\text{if } (*) \text{ goto } pc')}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc', l, h \,\rangle} \qquad [\textsc{IfGotoFalse}]\frac{c(pc) = (\text{if } (*) \text{ goto } pc')}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l, h \,\rangle}$$

$$[\textsc{Call}]\frac{\begin{array}{c} c(pc) = (x = y.m'(a_1, \ldots, a_n)) \\ (h', v_{\text{ret}}) = \text{Call}_h\,(l(x), m', l(a_1), \ldots, l(a_n)) \end{array}}{\langle\, pc, l, h \,\rangle \to_c \langle\, pc + 1, l\,[x \mapsto v_{\text{ret}}]\,, h' \,\rangle}$$

$$[\textsc{Return}]\frac{c(pc) = (\text{return } x)}{\langle\, pc, l, h \,\rangle \to_c \langle\, \text{end}(c), l\,[\mathsf{ret} \mapsto l(x)]\,, h \,\rangle}$$

$$(h', v_{\text{ret}}) = \text{Call}_h\,(r, m, v_1, \ldots, v_n) \equiv \begin{cases} c = \text{Lookup}\,(m, \text{Class}\,(r)) \\ l = [\mathsf{this} \mapsto r, \mathsf{arg}_1 \mapsto v_1, \ldots, \mathsf{arg}_n \mapsto v_n] \\ \langle\, 0, l, h \,\rangle \to_c^* \langle\, \text{end}(c), l', h' \,\rangle \\ v_{\text{ret}} = l'(\mathsf{ret}) \end{cases}$$

Figure 6: Semantics of Mini.

3.2.3 Android semantics

In this section, we define a semantics $(\cdot \rightsquigarrow \cdot) \in \mathbf{State} \to \mathbf{State}$ for the execution of an application. It defines what Android can do with the heap, the activity stack and the callbacks using the Call predicate defined in the previous section.

When the launch of an activity has been scheduled by StartActivity, it can be performed under a few conditions. The new activity is pushed on the stack and its initial state is uninit, only if the launching activity is in state stopped or visible:

$$[\textsc{NewActivity}]\frac{\begin{array}{c} h(r_a)(\mathsf{pending}) = cl :: p \\ h(r_a)(\mathsf{state}) \in \{\mathsf{stopped}, \mathsf{visible}\} \\ r_b \notin \mathrm{dom}\,(h) \quad \mathrm{Class}\,(r_b) = cl \quad o_b = [\mathsf{state} \mapsto \mathsf{init}] \end{array}}{\langle\, h,\, r_a :: as \,\rangle \rightsquigarrow \langle\, h\,[r_b \mapsto o_b]\,,\, r_b :: r_a :: as \,\rangle}$$

When the top activity is active, the user can press the back button and schedule its removal that will be performed by Remove:

$$[\textsc{Back}]\frac{\begin{array}{c} h(r_a)(\mathsf{state}) = \mathsf{active} \\ o'_a = h(r_a)\,[\mathsf{finished} \mapsto \mathbf{tt}] \end{array}}{\langle\, h,\, r_a :: as \,\rangle \rightsquigarrow \langle\, h\,[r_a \mapsto o'_a]\,,\, r_a :: as \,\rangle}$$

When the top activity is active, the user can trigger any listener of that activity. The call is performed by the Call construct which is defined in figure 6:

$$[\textsc{Click}]\frac{\begin{array}{c} h(r_a)(\mathsf{state}) = \mathsf{active} \\ (h', \_) = \mathrm{Call}_h\,(r_l, \mathrm{onClick}, r_a) \quad r_l \in h(r_a)(\mathsf{listeners}) \end{array}}{\langle\, h,\, r_a :: as \,\rangle \rightsquigarrow \langle\, h',\, r_a :: as \,\rangle}$$

Any finished activity in the destroyed state can be removed from the stack:

$$[\textsc{Remove}]\frac{h(r_a)(\mathsf{state}) = \mathsf{destroyed} \quad h(r_a)(\mathsf{finished}) = \mathbf{tt}}{\langle\, h,\, as_1 :: r_a :: as_2 \,\rangle \rightsquigarrow \langle\, h,\, as_1 :: as_2 \,\rangle}$$

Android can perform a move allowed by the lifecycle on any activity of the stack. To do this, it selects an activity, a target state and a method to apply according to **Lifecycle** defined in figure 7, invokes it and updates the state of the activity on the updated heap. There are two exceptions: when an activity is finished or when it has one or more pending activities:

$$[\textsc{Lifecycle}]\frac{\begin{array}{c} o'_a = h'(r_a)\,[\mathsf{state} \mapsto s'] \quad (h', \_) = \mathrm{Call}_h\,(r_a, m) \\ (s, m, s') \in \mathbf{Lifecycle} \quad s = h(r_a)(\mathsf{state}) \\ h(r_a)(\mathsf{finished}) = \mathbf{tt} \;\Rightarrow\; m \in \{\mathrm{onPause}, \mathrm{onStop}, \mathrm{onDestroy}\} \\ h(r_a)(\mathsf{pending}) \neq \epsilon \;\Rightarrow\; m \in \{\mathrm{onPause}, \mathrm{onStop}\} \end{array}}{\langle\, h,\, as_1 :: r_a :: as_2 \,\rangle \rightsquigarrow \langle\, h'\,[r_a \mapsto o'_a]\,,\, as_1 :: as_2 \,\rangle}$$

For the sake of simplicity, let us consider an example, an application whose code is given in figure 8. Suppose the manifest says that MainActivity is the activity to be launched when the user

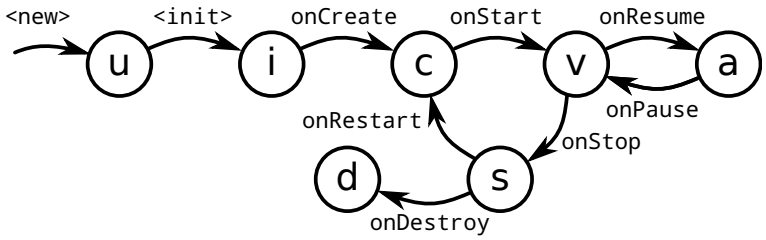

Figure 7: Definition of **Lifecycle** $\in \mathcal{P}(\textbf{ActivityState} \times \textbf{Method} \times \textbf{ActivityState})$ as an automata. $(s, m, s') \in$ **Lifecycle** means that there is a transition from state s to state s' by calling $a.m()$ where a is the activity on which the transition applies.

```
class MainActivity {
  method onCreate() {
    this.other = "OtherActivity";
    b = new Button();
    this.addListener(b)
  }
}
```

```
class Button {
  method onClick(a) {
    a.startActivity(a.other)
  }
}
```

```
class OtherActivity {
  method onCreate() {
    ...
  }
}
```

Figure 8: Sample application.

clicks the application's icon on the screen. Android can instanciate this activity, then apply a few LIFECYCLE rules to put it in the active state and will call method MainActivity.onCreate() in the process which will apply rule ADDLISTENER. As it is then active, the user can press the button just registered by onCreate. Android will then call method Button.onClick() on it. This method will schedule the launch of OtherActivity through STARTACTIVITY. When the callback is done, Android notices that its instance of MainActivity has a pending activity and can thus instanciate it. Then it can apply LIFECYCLE a few times to get the first activity stopped and this new one started. As a result, all the defined methods can be executed by our semantics.

3.3 On inferring the model and verifying its validity

Like any framework, the job of Android consists in manipulating the heap and the activity stack, and invoking the code of its applications. It is a big piece of software and understanding it is hard. However, one can infer the calling logic dynamically by logging calls to application's code. To do that, instrumentation is probably the best option but inserting logging statements into methods is not too difficult:

```
void onCreate(Bundle b) {
  Log.d("com.example.app", this + ".onCreate(" + b + ")");
  ...
}
```

This can give traces such as the following:

```
...
D/com.example.app( 1790): com.example.app.MainActivity.<clinit>
D/com.example.app( 1790): com.example.app.MainActivity@b1f49070.<init>()
D/com.example.app( 1790): com.example.app.MainActivity@b1f49070.onCreate(null)
D/com.example.app( 1790): com.example.app.MainActivity@b1f49070.onCreateLoader(0, null)
...
```

This kind of testing can give an idea of how Android works and can help verify the validity of the semantics. A much more sophisticated way of obtaining such results is to implement an interpreter and check it behaves the same way as the real system as is done in [19].

3.4 Conclusion

The semantics described here is approximate for two reasons. First, it does not model Android entirely as the real API provides many more methods. Second, it is probably more permissive than Android as the full state of the system is not taken into account. For instance, some semantic steps may be impossible under some conditions but we still allow them because it would take time to ensure such steps really cannot be taken by reading the source code and the resulting rules are more concise and manageable. Therefore, although the model cannot mimick exactly the real system, it can still give us sufficient insight on how to perform analyses on applications and design a proof of concept.

4 Analyzing Android

Applications are programmed using Java, an object-oriented language. A fundamental analysis issue in this kind of language is aliasing. This means that two different access paths can point to the same object. More precisely, when an object is modified from one reference, this modification will also be observable from any other reference to the same object. Therefore, an analysis such as taint analysis can have troubles determining the effect of some statements, which can in turn lead to a loss of precision. An advanced analysis aiming to tackle this problem is points-to analysis, also known as reference analysis or alias analysis [25]. In Android however, many objects such as Activities and Bundles are created or manipulated outside application code and passed as references to callbacks. We thus need to initialize points-to analysis with this information, which can be done with the model we have previously discussed.

4.1 Points-to analysis

Points-to analysis abstracts memory by points-to graphs representing what objects a variable or an object's field can point to. Naturally, as there can be an arbitrary number of objects at runtime, their locations are abstracted as well. We present here a traditional form of points-to analysis with an augmented Mini language by associating unique labels with every New statements. This way, objects can be abstracted by allocation sites:

$$x = \text{new}^h\ cl \quad \text{generates the following graph:} \quad [x \mapsto \{h\}]$$

The graph $L_m = [x \mapsto \{h\}]$ informally means that, in method m, variable x may point to objects allocated at $h \in \textbf{Site}$. Note that as a NEW statement can be executed several times, or never, we do not know how many real objects a site h can represent. We will consider points-to graphs $L_m \in \textbf{Variable} \to \mathcal{P}(\textbf{Site} \cup \textbf{String})$ for variables and $G \in \textbf{Site} \to \textbf{Field} \to \mathcal{P}(\textbf{Site})$ for fields. We will write $[x \mapsto P]$ for the points-to graph L_m such that $L_m(x) = P$ and $\forall v \neq x \quad L_m(v) = \emptyset$. Similarly, we will write $\left[h \xrightarrow{f} P\right]$ for the points-to graph G such that $G(h)(f) = P$ and $\forall h' \neq h \; \forall f \in \textbf{Field} \;\; G(h')(f) = \emptyset$. We have included constant strings in the points-to graphs, although it is not usually the case, to show that extensions can easily be made and this will be necessary in the following section. By convention, functions L_m and G are empty where they are not defined. For instance, for any string s, $\forall f \in \textbf{Field} \;\; G(s)(f) = \emptyset$.

To generate a points-to graph for a method, we first generate constraints for each statement. Once these constraints are defined, we solve them with an initial state and a fixpoint algorithm which computes the least solution. Returning to the NEW example, we have the following constraint:

$$[\text{CNEW}] \frac{h \in L_m(x)}{G, L \vdash x = \text{new}^h \; cl}$$

This rule CNEW means that for a solution (G, L) to be correct given a NEW, it must meet the contraint $h \in L_m(x)$. The constraints for MINI are described in figure 9. We assume the Lookup operator can determine the target method of a call.

Here is an example starting from $\text{CCall}_{G,L}(0, \text{foo})$, with the corresponding points-to graph:

```
method foo() {
  x = new1 A();
  y = new2 A();
  x.f = y;
  x = new3 B();
  y.f = x;
  z = x.bar()
}

method bar() {
  x = "in_bar";
  return this.f
}
```

$$L_{\texttt{foo}} = [\mathsf{this} \mapsto \{0\}, \mathrm{x} \mapsto \{1,3\}, \mathrm{y} \mapsto 2, \mathrm{z} \mapsto \{2\}]$$
$$L_{\texttt{bar}} = [\mathsf{this} \mapsto \{1,3\}, \mathrm{x} \mapsto \{\texttt{"in\_bar"}\}, \mathsf{ret} \mapsto \{2\}]$$
$$G = \left[1 \xrightarrow{f} \{2\}, 2 \xrightarrow{f} \{1,3\}, 3 \xrightarrow{f} \{2\}\right]$$

Note that we conclude that x may point to site 1 or 3 at the end of the program although it can only be site 3 as it has been overwritten. This comes from the fact that this analysis is flow-insensitive, which means that any instruction can be executed any number of times (including zero) and in any order. A flow-sensitive analysis, on the other hand, would take the control-flow into account and be more precise.

4.2 API

To define constraints for API calls, we need to extend the abstract domains that we introduced in the previous section to abstract **ActivityInfo** into **AbstractInfo**. Abstract values, as lattices, always have a smallest element and a greatest one, that is why we have added $\top$ and $\bot$ elements whenever necessary.

$$[\textsc{CString}]\frac{s \in L_m(x)}{G, L \vdash_m x = \texttt{"}s\texttt{"}} \qquad [\textsc{CNew}]\frac{h \in L_m(x)}{G, L \vdash_m x = \text{new}^h\ cl}$$

$$[\textsc{CAssign}]\frac{L_m(y) \subset L_m(x)}{G, L \vdash_m x = y}$$

$$[\textsc{CSet}]\frac{\forall h \in L_m(x)\ \ L_m(y) \subset G(h)(f)}{G, L \vdash_m x.f = y} \qquad [\textsc{CGet}]\frac{\forall h \in L_m(y)\ \ G(h)(f) \subset L_m(x)}{G, L \vdash_m x = y.f}$$

$$[\textsc{CCall}]\frac{\forall h \in L_m(y)\ \ \text{CCall}_G\,(h, m', L(a_1), \ldots, L_m(a_n)) \subset L_m(x)}{G, L \vdash_m x = y.m'(a_1, \ldots, a_n)}$$

$$[\textsc{CReturn}]\frac{L_m(x) \subset L_m(\mathsf{ret})}{G, L \vdash_m \text{return } x}$$

$$\text{CCall}_{G,L}\,(h_0, m, H_1, \ldots, H_n) \subset H_{ret} \equiv \begin{cases} c = \text{Lookup}\,(m, \text{Class}\,(h_0)) \\ h_0 \in L_m(\mathsf{this}) \\ H_1 \subset L_m(\mathsf{arg_1})\ \ \ldots\ \ H_n \subset L_m(\mathsf{arg_n}) \\ G, L \vdash_{m'} c \\ L_m(\mathsf{ret}) \subset H_{ret} \end{cases}$$

$$[\textsc{CCode}]\frac{\forall s \in c\ \ G, L \vdash_m s}{G, L \vdash_m c}$$

Figure 9: Points-to constraints for Mini.

$$[\textsc{CStartActivity}]\frac{\forall h \in L_m(x)\ \ L_m(y) \sqsubset G(h)(\mathsf{pending})}{G, L \vdash_m x.\mathrm{startActivity}(y)}$$

$$[\textsc{CFinish}]\frac{\forall h \in L_m(x)\ \ \mathbf{tt} \sqsubset G(h)(\mathsf{finished})}{G, L \vdash_m x.\mathrm{finish}()}$$

$$[\textsc{CAddListener}]\frac{\forall h \in L_m(x)\ \ L_m(y) \sqsubset G(h)(\mathsf{listeners})}{G, L \vdash_m x.\mathrm{addListener}(y)}$$

$$[\textsc{CSetListener}]\frac{\forall h \in L_m(x)\ \ L_m(y) \sqsubset G(h)(\mathsf{listeners})}{G, L \vdash_m x.\mathrm{setListener}(y)}$$

Figure 10: Constraints for API methods.

$$\begin{aligned}\mathbf{AbstractInfo} = \{&\\ &\mathsf{state} : \mathbf{ActivityState}^{\top}_{\bot},\\ &\mathsf{pending} : \mathcal{P}(\mathbf{Class}),\\ &\mathsf{listeners} : \mathcal{P}(\mathbf{Site}),\\ &\mathsf{finished} : \mathbf{Boolean}^{\top}_{\bot}\\ \}&\end{aligned}$$

In **AbstractInfo**, state represents the possible state of an activity object. It can be an exact state $\in$ **ActivityState**, or $\top$ if it is unknown. It is $\bot$ if the object is not an activity or the attribute is undefined. pending contains potential activity names to be launched. Attribute listeners contains the possible sites in the listeners attribute of the concrete **ActivityInfo**. Similarly, finished contains potential values of the concrete attribute. The analysis domain is extended with the new abstract values as follows:

$$\begin{aligned}\mathbf{AbstractValue} &= \mathcal{P}(\mathbf{Site} \cup \mathbf{String})\\ L_m \in \mathbf{Local} &= \mathbf{Variable} \to \mathbf{AbstractValue}\\ G \in \mathbf{Global} &= (\mathbf{Site} \to \mathbf{Field} \to \mathbf{AbstractValue}) \times (\mathbf{Site} \to \mathbf{AbstractInfo})\end{aligned}$$

We give the constraints related to API methods in figure 10. Note that given the flow-insensitivity of the analysis, rules CAddListener and CSetListener are the same.

4.3 Application

This section explains how executions of an application are abstracted. The concrete state contains a heap and an activity stack while the **Global** abstract domain we considered only abstracts the heap. Let us abstract the stack by $\mathbf{AbstractStack} = \mathcal{P}(\mathbf{Site})$, the potential sites of Activity classes on the stack. We now want to give constraints to $A, G, L \vdash app$ where $(A, G, L) \in \mathbf{AbstractStack} \times \mathbf{Global} \times (\mathbf{Method} \to \mathbf{Local})$.

Let us suppose we have a function next that for each abstract state (A, G) computes a set of call sites of the form $(h, m, H_1, \ldots, H_n)$ where h is a receiver site, m is a method to apply and

$H_1, \ldots, H_n$ are sets of sites taken as arguments. It represents the potential actions Android may take from a given state. This lets us have the following rule:

$$[\text{CApp}]\frac{\forall (h, m, H_1, \ldots, H_n) \in \mathsf{next}\,(A, G) \quad \begin{cases} \text{CCall}_{G,L}\,(h, m, H_1, \ldots, H_n) \in \_ \\ \text{CLifecyle}_G\,(h, m) \sqsubset G \end{cases}}{A, G, L \vdash app}$$

where:

$$\text{CLifecycle}_G\,(h, m) = G \text{ with potential next state for } h \text{ according to } m \text{ and } \textbf{Lifecycle}$$

If we execute this analysis on the application from figure 8 with the Button allocation having the label 1 and OtherActivity.onCreate containing the single instruction `s = "secret"`, we get the following solution where α and β refer to the allocation sites of MainActivity and OtherActivity respectively:

$$\begin{aligned} A &= \{\alpha, \beta\} \\ G(\alpha) &= [\mathsf{state} \mapsto \top, \mathsf{listeners} \mapsto \{1\}, \mathsf{pending} \mapsto \{\beta\}, \mathsf{finished} \mapsto \top, \text{other} \mapsto \texttt{"OtherActivity"}] \\ G(\beta) &= [\mathsf{state} \mapsto \top, \mathsf{finished} \mapsto \top] \\ G(1) &= [] \\ L_{\text{MainActivity.onCreate}} &= [\mathsf{this} \mapsto \{\alpha\}, \mathsf{b} \mapsto \{1\}] \\ L_{\text{OtherActivity.onCreate}} &= [\mathsf{this} \mapsto \{\beta\}, \mathsf{s} \mapsto \texttt{"secret"}] \\ L_{\text{Button.onClick}} &= [\mathsf{this} \mapsto \{1\}, \mathsf{a} \mapsto \{\alpha\}] \end{aligned}$$

From this information, we can draw a similar conclusion as in section 3.2.3, that is, every method can be executed and if there were a kind of `sink(s)` instruction at the end of OtherActivity.onCreate, we could raise an alarm about a potential leak. But we are more general than taint analysis because we have other information that can be useful for other analyses. However, the result is quite unprecise regarding how Android executed the application. We have not much information about the states or what user actions could have led to the execution of the methods (of course, this is quite obvious for this simple application, but would not be for a much larger one).

4.4 Improving precision

To make the previous results more useful and closer to real executions, we consider two improvements. We first consider the potential gains from adding flow-sensitivity, which is a classical feature of static analysis. We then design a custom context-sensitive analysis based on our semantics of Android.

4.4.1 Flow-sensitivity

Given the semantics of our language, statements in a method are executed in a specific order. For instance, the methods of the sample application are implemented with straight-line code. Therefore, there is a guarantee that statements in such code are executed once and in order, which can give us

more precise information. However, this order is generally not computable because of the IfGoto statement but we can over-approximate it by considering both IfGotoTrue and IfGotoFalse to be possibly taken at each such statement.

A method can be seen as a control-flow graph, a directed graph linking statements together according to straight-line ordering and IfGoto statements. A flow-sensitive analysis computes an abstract state at each vertex, or each program point, instead of globally. Constraints specify how an abstract state is transfered through statements and edges. Because of IfGoto, some program points can be reached from more than one edge. In such a case, abstract states coming from those edges are merged.

For instance, an instruction $y = x$ would transform $[x \mapsto \{1\}, y \mapsto \{2\}]$ into $[x \mapsto \{1\}, y \mapsto \{1\}]$ instead of $[x \mapsto \{1\}, y \mapsto \{1, 2\}]$ if it were flow-insensitive. However, the same cannot be done for fields because, for example, in state $\left[1 \xrightarrow{f} 2\right]$, site 1 can abstract several real objects and an assignment to its field f generally cannot be guaranteed to affect all potential instances. This is called a weak update. An analysis with more precise abstractions can still perform strong updates for fields when it determines the allocation site under consideration can have at most one instanciation, which is the case for many objects in Android.

A flow-sensitive analysis is more expensive than a flow-insensitive one because it computes abstract states at each program point instead of globally. However, it can improve precision because we can choose better analysis domains for Android and perform strong updates. For instance, the finished field of an Activity could be computed precisely after a callback if the analysis can make sure a finish() call has been issued.

4.4.2 Context-sensitivity

Even with flow-sensitivity, precision can be lost when there are several sequences of calls that can lead to the execution of a callback. For instance, if Button.onClick were potentially active in different situations with different activity stacks, it would be interesting to see a difference in the result of the analysis.

In order to take such contexts into account, the analysis must compute different solutions for each of them and transfer abstract states from one context to successor contexts determined by next, CCall and CLifecycle, the same way a flow-sensitive analysis transfers states between program points via flow edges.

The question of which contexts to choose depends on the precision required from the analysis. Having only one context is equivalent to applying the analysis from section 4.3. A sensible choice is to differentiate contexts based on the full Android state (**AInfo** and **AbstractStack**). This takes advantage of our semantics by relying on its precision. With this improvement, we believe we could get results with precise Android attributes and an idea of which actions can lead to reachable states.

Combined with flow-sensitivity, a context-sensitive analysis could output the contexts in figure 11 for the sample application. As can be seen, context c_2 has more than one incoming callbacks, which means that incoming abstract states are merged at this point. If this is undesired, contexts can be refined to differentiate the two cases. For each context c, the analysis computes a solution

$$
c_0 = \left(\begin{array}{rl} A &= \{\alpha\} \\ G(\alpha) &= [\mathsf{state} \mapsto \mathsf{uninit}, \mathsf{listeners} \mapsto \emptyset, \dots] \end{array} \right)
$$
$$
\downarrow (\alpha, \text{<init>})
$$
$$
\vdots
$$
$$
\downarrow (\alpha, \text{onCreate})
$$
$$
c_2 = \left(\begin{array}{rl} A &= \{\alpha\} \\ G(\alpha) &= [\mathsf{state} \mapsto \mathsf{init}, \mathsf{listeners} \mapsto \{\{1\}\}, \dots] \end{array} \right)
$$
$$
\downarrow (1, \text{onClick}, \{\alpha\}) \qquad \uparrow (\alpha, \text{onRestart})
$$
$$
\vdots \qquad\qquad \vdots
$$

Figure 11: Partial contexts for the application from figure 8.

$(A(c), G(c))$ which is more precise than without contextes.

4.5 Implementation

In order to experiment with our semantics and context-sensitivity, we implement an analyzer in the OCaml language for applications written in Mini. The analyzer has three main parts: a parser to convert applications such as the one from figure 8 into an abstract syntax tree, a context-sensitive Android analysis based on rule CApp and an intra-callback analysis based on rules from figures 9 and 10.

The AST contains both a set of classes and a manifest. The latter enables the analyzer to determine how the application can be initialized. In this thesis, we have only considered launching an application by clicking its icon on the main screen but there are other possible initializations on a real system.

The Android analyzer first converts the AST into a format that can be more easily manipulated. For each method, the corresponding sequence of instructions is converted to a control-flow graph of type $(label \times \mathcal{P}(label \times inst \times label) \times label)$ which an initial label and a final one, and where $inst$ is the set of elementary instructions (without IfGoto).

It then computes an initial state based on the manifest and solves the Android constraints using a simple fixpoint algorithm:

```
(* fixpoint : ('a -> 'a -> bool) -> ('a -> 'a) -> 'a -> 'a *)
let fixpoint eq transfer l0 =
  let rec iter l =
    let l' = transfer l in
    if eq l l' then
      l (* this is the solution *)
    else
      iter l' (* continue iterating *)
in
iter l0 (* iterate from the initial state *)
```

The analyzer must provide the `fixpoint` function with an equality function and a transfer function. This makes the algorithm generic and not specific to a particular analysis.

The Android transfer function encodes the rule CApp by implementing CCall and CLifecycle. CCall needs an intra-callback analysis which uses the fixpoint algorithm again to compute a solution for the control-flow graph of the analyzed callback.

All the abstract domains, or lattices, have been implemented as separate modules, each providing common operations for lattices such as `equal`, `join` and `bot`. For instance, the operations on local environment lattices are defined from the operations on value lattices which in turn are defined from other lattices.

We hope that this approach will enable us to adapt our implementation to easily test potential improvements of our analysis. This is still a work in progress but the implementation can already analyze small Mini applications with context-sensitivity.

Conclusion

We have extended existing models of Android to abstract more features of the framework: the lifecycle of Activities, the control-flow logic of the system and the dynamic registration of callbacks. This enabled us to adapt points-to analysis to applications and determine that it must be extended to provide meaningful information about applications. Our own extension is a form of context-sensitivity aimed at capturing the relevant states of Android to improve precision and return a more detailed analysis result. Additionally, the sensitivity of our analysis can be parameterized so that based on how it is used, a acceptable tradeoff between precision and cost can be determined.

Many features of applications have been ignored to focus on those we considered to be essential to Android. However, now these first obstacles are overcome, it would be interesting to consider features such as reflexivity or Uris. The implementation should be modular enough to easily provide proof-of-concepts for future evolutions of our analysis.

We believe our approach can benefit several analyses. It can be envisioned to serve as a basis for taint analysis, vulnerability discovery and bug finding. Another use we could think of is as a mean of automating offline dynamic analyses that use precise runtime information but need operators to run the applications under analysis.

References

[1] M. Mitchell, J. Oldham, and A. Samuel, *Advanced Linux Programming*. New Riders Publishing, 2001.

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and comm. security*, p. 627638, 2011.

[3] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX Conference on Security*, SEC'11, (Berkeley, CA, USA), p. 21, 2011.

[4] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 6, 2012.

[5] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component Communication Mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, p. 239252, 2011.

[7] P. McDaniel and W. Enck, "Not so great expectations: Why application markets haven't failed security," *Security & Privacy, IEEE*, vol. 8, no. 5, p. 7678, 2010.

[8] T. Vidas, D. Votipka, and N. Christin, "All Your Droid are Belong to Us: A survey of current Android attacks.," in *WOOT*, p. 8190, 2011.

[9] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Network & Distributed System Security Symposium*, NDSS'13, 2013.

[10] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android Applications," *Manuscript, Univ. of Maryland*, 2009.

[11] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *Proceedings of the Workshop on Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2012.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.," in *OSDI*, vol. 10, p. 255270, 2010.

[13] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "Highly Precise Taint Analysis for Android Applications," Tech. Rep. TUD-CS-2013-0113, EC SPRIDE, May 2013.

[14] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," in *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *PLDI*, p. 29, 2014.

[16] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for Java," in *Programming Languages and Systems*, p. 139160, Springer, 2005.

[17] E. R. Wognsen and H. S. Karlsen, "Static Analysis of Dalvik Bytecode and Reflection in Android," 2012. M. Sc. thesis, Aalborg University.

[18] J. Jeon, K. K. Micinski, and J. S. Foster, "SymDroid: Symbolic Execution for Dalvik Bytecode," Tech. Rep. CS-TR-5022, Department of Computer Science, University of Maryland, College Park, July 2012.

[19] E. Payet and F. Spoto, "An Operational Semantics for Android Activities," in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, (New York, NY, USA), pp. 121–132, ACM, 2014.

[20] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *SOAP@PLDI*, pp. 1–6, 2014.

[21] A. Rountev and D. Yan, "Static Reference Analysis for GUI Objects in Android Software," in *CGO*, p. 143, 2014.

[22] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *ACM Conference on Computer and Communications Security*, pp. 1043–1054, 2013.

[23] L.-K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *USENIX Security Symposium*, pp. 569–584, 2012.

[24] S. N. Freund and J. C. Mitchell, "A Formal Framework for the Java Bytecode Language and Verifier," in *OOPSLA*, pp. 147–166, 1999.

[25] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 1–41, Jan. 2005.